5-2014

A Standard Network Adapter for Spacecraft and Payloads

Colby Russell Salmon
Utah State University

A STANDARD NETWORK ADAPTER FOR SPACECRAFT AND PAYLOADS

by

Colby Russell Salmon

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Curtis Dyreson Major Professor	Dr. Daniel Watson Committee Member

Dr. Jacob H. Christensen
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

A Standard Network Adapter for Spacecraft and Payloads

by

Colby Russell Salmon, Master of Science

Utah State University, 2014

Major Professor: Dr. Curtis Dyreson
Department: Computer Science

SNAP reduces space mission cost by providing an advanced modular, open systems approach to software. As part of the space modernization efforts, SMC is sponsoring an effort to develop advanced network architectures for spacecraft systems. Government and DoD agencies are looking to the hosted payload concept to reduce space mission cost in the current budget constrained environment. The hosted payload paradigm poses significant system integration challenges due to payloads being developed independently of the host spacecraft. SNAP is an adapter that uses software and hardware interfaces to provide network connectivity and protocol translation to integrate a spacecraft and payload. In addition to solving the systems integration problems associated with the hosted payload concept, SNAPs advanced modular, open systems approach results in reduced overall cost and length of development for spacecraft systems. SNAP has been demonstrated with the Operationally Responsive Space (ORS) offices Modular Space Vehicle (MSV) and three different payloads. This document provides a system level overview of the SNAP architecture and the design concepts used in its development.

(44 pages)

ACKNOWLEDGMENTS

This report would not be complete without acknowledging the help and encouragement I received during its creation. I would also like to take this opportunity to acknowledge the support and assistance that I was given during the broader effort of completing the Master of Science in Computer Science degree program at Utah State University.

- Space Dynamics Laboratory, for providing opportunities and supporting my academic goals.
- Jacob Christensen, for giving me a chance and mentoring me in computer science.
- Taryn, for her endless patience. Without her insight, I would not have been introduced to computer science. Without her support, I would not have the luxury of pursuing it.

This report is based upon work supported by the United States Air Force under Contract No. FA8814-09-D-0001, Task Order 1, Subtask SNAP. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Air Force.

CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF FIGURES vii

ACRONYMS viii

CHAPTER

1 INTRODUCTION 1
- 1.1 Overview 1
- 1.2 Background 1
- 1.3 Related Work 3

2 DESIGN 4
- 2.1 Overview 4
- 2.2 Concept 4
- 2.3 Scope 5

3 MONA 7
- 3.1 Modular Open Network Architecture 7
- 3.2 Layered Architecture 10
- 3.3 Competitive and Proprietary in a MONA System 12

4 SNAP ARCHITECTURE 13
- 4.1 SNAP Core Software Architecture 13
- 4.2 Infrastructure 13
- 4.3 Services 14
- 4.4 Translation 19

5 SYSTEM INTEGRATION 23
- 5.1 Integrating a Spacecraft Bus 23
- 5.2 Integrating a Payload 24

6 SNAP IMPLEMENTATION 26
- 6.1 Spacecraft Bus Translator Implementation 26
- 6.2 Payload Translator Implementation 29

7 CONCLUSION 33
- 7.1 Results 33

REFERENCES 35

LIST OF FIGURES

Figure		Page
2.1	SNAP translates between a payload and a spacecraft bus and incorporates MONA principles within the SNAP implementation	4
2.2	MONA native to payload and spacecraft	5
3.1	Modular software reduces cost by reducing re-testing	8
3.2	GAO analysis of DoD and industry data	9
3.3	Network model comparison	11
3.4	Proprietary components can interface to the MONA network	12
4.1	SNAP architecture	14
4.2	Payload sends data to ground via File and Data Service	16
4.3	Ground sends timed command stack	17
4.4	Ground sends command to payload	18
4.5	A spacecraft bus is adapted to SNAP a single time and is then compatible with any number of SNAP-adapted payloads	20
4.6	The Bus Translator handles command destined for spacecraft bus	21
4.7	The Payload Translator requests data from spacecraft bus	22
5.1	Spacecraft bus translation interface	24
5.2	Payload translation interface	25
6.1	Each derived bus translator must implement the pure virtual functions of the parent bus translator	27
6.2	Each derived payload translator must implement the pure virtual functions of the parent payload translator	30

ACRONYMS

AIAA American Institute of Aeronautics and Astronautics

API Application Programing Interface

CHIRP Commercially Hosted Infra-Red Payload

DISC Digital Imaging Space Camera

DoD Department of Defence

GAO United States Government Accountability Office

MONA Modular Open Network Architecture

MSV Modular Space Vehicle

ORS Operationally Responsive Space

OSI Open Systems Interconnect

SMC Space and Missile Systems Center

SMC/XR Space and Missile Systems Center Development Planning Directorate

SNAP Standard Network Adapter for Payloads

SOFIE Solar Occultation For Ice Experiment

SPA Space Plug-and-play Architecture

SSM SPA Services Manager

CHAPTER 1

INTRODUCTION

1.1 Overview

This report provides a description and system level overview of the Standard Network Adapter for Payloads (SNAP). Motivation for its development and the benefits of using SNAP will also be explained. SNAP is a system comprised of both software and hardware for connecting potentially disparate electrical and data protocol interfaces between a host spacecraft and payloads. This report focuses primarily on the software.

The SNAP software provides services and management of networked communications (data and commands) with a spacecraft bus and payload(s), as well as internal communications within the SNAP host avionics. SNAP includes general purpose processing; data routing (enabling multiple payloads per SNAP unit); protocol translation; time and ephemeris distribution from the host to the payloads; and data buffering to assist in store and forward scenarios.

The development of SNAP was sponsored by the US Air Force Space and Missile Systems Center Development Planning Directorate (SMC/XR). SNAP was developed in collaboration with other Modular Open Network Architecture (MONA) efforts for spacecraft including; DARPA F6, NASA CII, AFRL Monarch, and ORS MSV.

1.2 Background

The current industry practice for satellite system development is an expensive and time consuming process. Due to the unique system requirements for each particular mission, satellite systems are typically build to order. Creating each new custom satellite generates new, complex interfaces, and requires significant systems engineering followed by costly

testing and verification.

The US Department of Defense (DoD) is working to reduce acquisition costs as a way to increase operational capabilities while complying with federal budget constraints [1]. They have also recognized the importance of reducing the development time in order to respond in a timely manner to changing operational needs, and have emphasized compliance with acquisition policies in regard to open systems [2].

The DoD and other agencies within the U.S. Government are also looking to the hosted payload concept to achieve their cost and schedule goals. Gen. William Shelton, commander of Air Force Space Command, has said the Defense Department must consider hosted payloads as one way to help keep costs down [3].

The idea of commercially hosted payloads has been validated by the CHIRP program. The CHIRP program successfully accomplished its mission objectives which involved an experimental infrared sensor being hosted on a commercial communications satellite. This program saved 85% of the estimated $1.2 billion it would have cost to complete the mission on a dedicated satellite.

Taking the lessons learned from CHIRP, integrating a satellite with a payload that was developed independently poses engineering challenges. Consider that the payloads in question typically take years longer to develop than commercial satellites. While commercial satellite companies have been willing to support the idea of hosting payloads, they are primarily concerned with the schedule of their main mission, and will not adjust the schedule to wait for a payload. This is a problem because a payload needs to be developed independently and then integrated with any available spacecraft. This means the payload developers wont necessarily have all the information about the hardware and software interfaces of the host spacecraft.

One way to solve this problem is to have standardized hardware and software interfaces for spacecraft components to ensure native compatibility. While this is a desirable solution, industry standardization takes considerable time and effort.

1.3 Related Work

Several spacecraft interface technology initiatives are in development across the U.S. Government that may in part or in combination offer strategies to effectively deploy and utilize hosted payloads. SNAP offers a near-term solution for hosted payloads in a way that furthers the concept of open systems and standardized interfaces.

The United States Government Accountability Office (GAO) has endorsed the open systems approach and reported that reduced schedule and reduced life-cycle cost are among the benefits of open systems [4].

SMC/XR has the responsibility of developing architectures to support mission requirements within the SMC portfolio. A critical input to this process is analyzing and developing future concepts. SMC/XR is pursuing Modular Open Network Architecture (MONA) frameworks as a way to create affordable and resilient solutions to DoD acquisition needs [5]. As part of this effort, SMC/XR directed Space Dynamics Laboratory to develop SNAP using MONA principles; utilizing advanced communications network interfaces to support communications on hosted payloads.

The author, a software engineer at Space Dynamics Laboratory, was a member of the SNAP development team and wrote the *SNAP Software Information Guide* [6]. Much of the material in this report is based on the *SNAP Software Information Guide*. Other team members have published papers on the results of the SNAP project. The SMC Chief Scientist Dr. Roberta M. Ewart and SMC's Program Manager for SNAP Lt. Garrett W. Ellis authored several of these papers [5] [7].

CHAPTER 2

DESIGN

2.1 Overview

SNAP was designed as a technology demonstration that addresses the bus-payload integration problem, and acts as both a proof of concept and a sample implementation for U.S. Government agencies and their industry partners. The idea of SNAP is to drastically reduce the resources necessary to integrate busses and payloads by adapting incompatible interfaces; both hardware and software, in a way that minimizes or eliminates the need to modify the existing spacecraft and payload. SNAP is intended for use with payloads and spacecraft busses that may be incompatible due to independent or proprietary development; such as the hosted payload model, or an orphaned system in need of a replacement host or payload due to a component that fails integration and validation.

2.2 Concept

In order to meet the objective of minimal resources for system integration, SNAP provides an API. To integrate a payload to a spacecraft bus, the bus and the payload are both adapted to the standardized interface of SNAP with translator applications using the API. A translator application is executed on the SNAP hardware for each payload and the

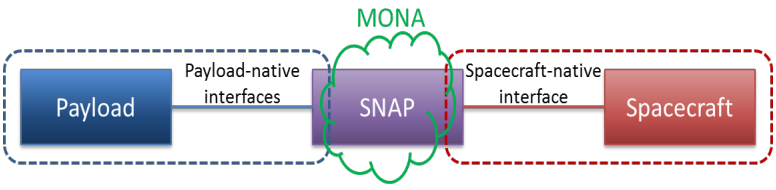


Figure 2.1. SNAP translates between a payload and a spacecraft bus and incorporates MONA principles within the SNAP implementation

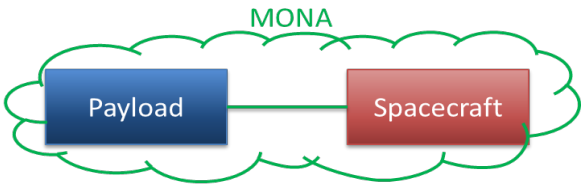


Figure 2.2. MONA native to payload and spacecraft

bus to be used with the system. Writing this single SNAP application for each component is a non-recurring effort; once this translator application is written, the component is SNAP compatible, and ready for use with any other SNAP compatible component. This means that a SNAP compatible bus can be used with a number of SNAP compatible payloads in a plug-and-play fashion. Use of an additional payload is accomplished by writing the SNAP application for that specific payload, with no further changes to the bus or its SNAP application. The fact that the bus and the payload are translated and adapted to a standard interface inside SNAP is transparent to either unit.

The ease of integrating spacecraft and payloads could be incorporated into traditional payload/spacecraft interfaces to reduce the cost and schedule associated with traditional programs. The longer term goal would be systems where the spacecraft and payload can natively communicate across a standardized MONA interface, as in Figure 2.2.

2.3 Scope

The SNAP software is modular in nature and is implemented as a suite of applications. The suite leverages the core components of the SPA Services Manager (SSM) [8], including the SSM API. SNAP and SSM adhere to the Space Plug-and-Play Architecture (SPA) standards [9] approved by the American Institute of Aeronautics and Astronautics (AIAA). Software and hardware implementation for system interconnection using RS-422 and SpaceWire were completed for SNAP.

SNAP by itself is not a turn-key solution to a MONA approach across the entire mission. SNAP does not address networked communications from space to ground, nor networked ground links. SNAP has been implemented in a step-wise fashion in order to demonstrate

technology capabilities and value, and requires additional work before it is flight ready. Additional software modules will be necessary to increase functionality, such as encryption or interconnection using other physical media.

CHAPTER 3

MONA

3.1 Modular Open Network Architecture

SNAP incorporates MONA principles, which stands for Modular Open Network Architecture. The Modular aspect of MONA means that the software is designed (in part), by separation of the software responsibilities into different functional units, or modules. These individual software components are used as building blocks to create a larger, complete software system. Modules can be aggregated in various ways to provide the flexibility to meet different system requirements.

Modular software has many advantages when designed and implemented well. Separation of the software concerns results in the modules being decoupled. This means that changes in one portion of a system remain isolated, and do not propagate or have far-reaching effects in other parts of the system. Proper execution of modularity results in software that is more maintainable because the components are capable of being upgraded individually to respond to changing requirements or to increase functionality. This limits the cost and scope of software upgrades. Software testing and validation is also simplified because the system can be decomposed back into individual functional units. The individual functional units are easier to test.

Modularity also results in decreased testing costs by increases the reusability of software. All software is reusable to some extent. We often say that roughly 80% of software is reused from one program to another. Unfortunately, that typically means that 80% of each software component is reused, while the remaining 20% requires changes to adapt to new hardware. In the old paradigm, introducing new hardware for a new mission means the interfaces between hardware and software change. This is costly because the software

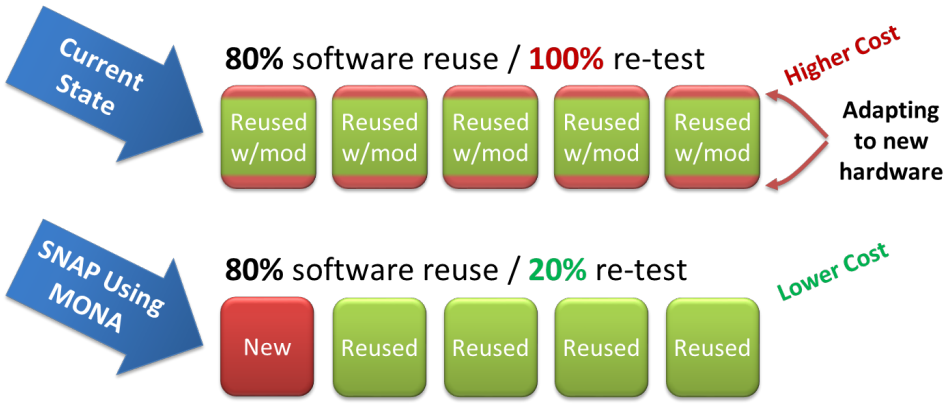
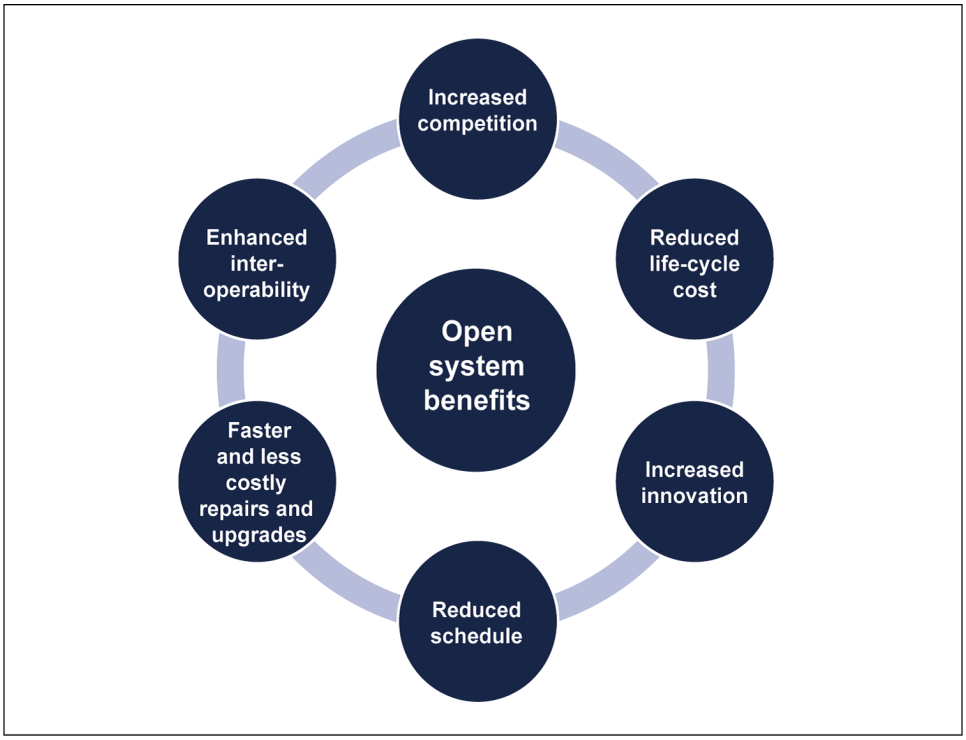


Figure 3.1. Modular software reduces cost by reducing re-testing

that was only 20% modified needs to be 100% re-tested. SNAP represents a paradigm shift because it has been designed and implemented in such a way that modules are reused in their entirety. Using this modular architecture improvement, 80% software reuse and 20% new or modified software for a space mission results in only 20% re-test. The modules that are reused intact do not need to be re-tested.

One of the keys to modularity is standardized interfaces. Modules that are written to the same interface are interchangeable and swappable. This is an important concept; two modules may have completely different internal implementations to accomplish the same task (interchangeable), or be implemented to accomplish the task with different hardware or protocols (swappable), as long as they written to the same interface. This type of interface programming is facilitated by having standardized interfaces.

The Open aspect of the MONA acronym indicates that the standards are open and available to everyone. The benefits of open systems are discussed at length in a report by the GAO [4]. The GAO report lists product manufacturers, consumers, and industries as recipients of these benefits because open systems spur growth, competition, and innovation. Building on an infrastructure that is supported by the industry increases the knowledge base of the infrastructure, reduces risk, and lowers the costs of product development. Increased competition puts more focus on product value, performance, and development schedules. Figure 3.2 comes from the GAO report and is used here as a concise way to highlight some



Source: GAO analysis of DOD and industry data.

Figure 3.2. GAO analysis of DoD and industry data

of the reports findings.

MONA requires standardized interfaces, protocols, and processes. Having the standards open is the key to everyone working to the same standard, so that the individual software components are truly modular. With an open standard, individual components can be built without prior knowledge of how the system will be constituted in its final form. It is important to note that an open standard does not preclude using proprietary implementations. This concept is further discussed in section 3.3.

The Network part of MONA is message-based, interconnected communications. The modular and open standards concepts are used to build an infrastructure for networked communications. The network infrastructure increases the modularity of the system and decreases the complexity of a module by eliminating the need of a single module to know how to communicate with every other module. Instead, each module just communicates with the network infrastructure which then provides data movement and delivery as a service to

all of the modules.

These concepts do not suggest scrapping all the software the aerospace industry has already built, only changing how the software works together and with the hardware so that we improve the reusability and build on a common infrastructure.

MONA is about building a network communications infrastructure in such a way as to make it reusable and extensible to future interconnection networks. Reusing a standardized infrastructure reduces system development time and cost, and frees up resources for increased content and capability rather than having individual entities generate independent infrastructures.

3.2 Layered Architecture

In addition to the MONA principles, SNAP is also a layered system adhering to a simplified version the Open Systems Interconnect (OSI) model [10]. The OSI model is a broadly recognized and industry standard means of providing an ordered, flexible, and extensible communications system architecture. This is how billions of lines of code have been written for terrestrial network infrastructures such as cell phones communications and the internet.

The layers of the OSI model promote easier understanding of the architecture. The design of each layer can be addressed individually, reducing the complexity of the associated communication system. Each layer can be developed independently, can be tested at the layer boundaries, and is easier to maintain. Since the responsibilities of the layer are well categorized, swapping out a layer with an alternate implementation has minimal impact on other layers. This provides a well-defined methodology for adding new and future network technologies without affecting existing components. For example, support for a new or future interconnection medium can be added by creating a new software module at the data link layer.

For its network infrastructure, SNAP make use of the SPA Services Manager (SSM) software. The SSM is an implementation of the AIAA SPA standards including the SPA Networking Standard. The SSM was originally developed for Air Force Research Laboratory

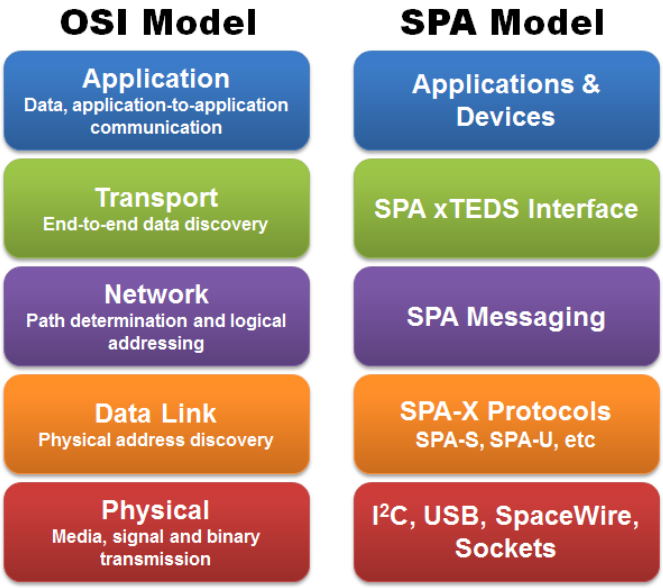


Figure 3.3. Network model comparison

by SDL using an ISO 9001 certified spaceflight software development process. SPA and SSM are being actively used for the ORS/MSV. Full documentation of the SSM is available to registered persons at https://pnpsoftware.sdl.usu.edu/projects/ssm/wiki.

The SPA model is fundamentally a simplified form of the OSI model, using a hierarchy of five abstraction levels, with the traditional session and presentation layers being included in the Application layer. The layers in the architecture model for SPA are the Physical, Data Link, Network, Transport, and Application layers (see Figure 3.3).

A layered architecture is a further refinement of modularity. In addition to components encapsulating and addressing a different aspect of the communication system requirements, each layer is only required to interact with the layers directly above and below it. These layers represent a hierarchy of abstraction levels, where a given layer provides services to higher level layers and receives services from the layer below. The result is that software components interoperate in a plug-and-play manner at the Application layer. Furthermore, the components communicate and interoperate independently of the nature of the resource. Consumers of services remain unaware of the physical type or the physical network location of the component providing the service.

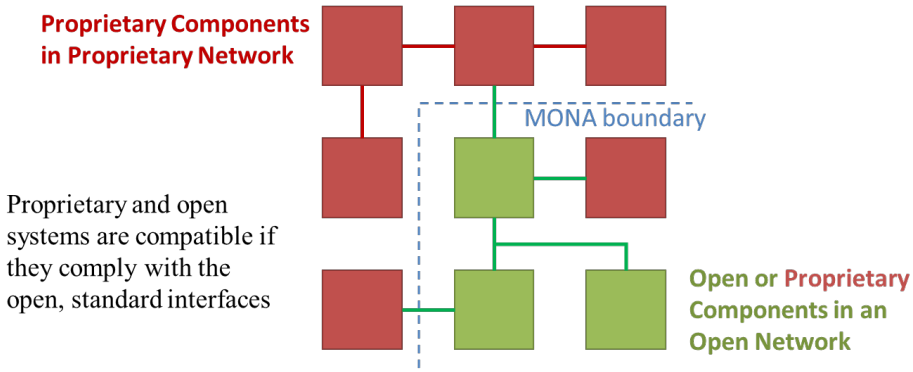


Figure 3.4. Proprietary components can interface to the MONA network

3.3 Competitive and Proprietary in a MONA System

It is important to note that SNAP and MONA do not preclude proprietary software. Compatibility can be achieved by exposing a standardized interface, while the implementation remains private. The PC industry has demonstrated that standardized communications between computers does not prohibit competition or proprietary implementations of computers and software. A good example of this can be found with email clients. While there are many free and/or open implementations of email clients (e.g. Gmail, Thunderbird), there are also very successful proprietary implementations (e.g. Microsoft Outlook, Apple Mail).

Under the MONA paradigm, standardized communications for space systems represents a network infrastructure as an industry shared investment, with shared benefit. Competition is encouraged on cost, performance, and proprietary capabilities rather than proprietary infrastructure.

CHAPTER 4

SNAP ARCHITECTURE

4.1 SNAP Core Software Architecture

The SNAP software is architected in three groups or tiers. The three tiers are; Infrastructure, Services, and Translation. In the context of the OSI model, the components of the Translation and Service tiers of SNAP reside in the Application layer. The communication and interoperation of these applications is supported by the SNAP Infrastructure tier, which represents the other four OSI layers; Transport, Network, Data Link, and Physical.

The Translation tier is the only tier that is updated when integrating a new payload or spacecraft bus. The Services and Infrastructure tiers are designed to remain unchanged as various payloads and busses are integrated. This reduces the time and cost of testing and validation because the reused tiers only need to be tested and validated once.

4.2 Infrastructure

The Infrastructure tier is responsible for the network communications on the SNAP processor and its peripherals. This tier is the N and part of the O in MONA, and is common to all payloads and spacecraft busses. The Infrastructure tier is comprised of three applications; Lookup Service, Central Address Service, and SPA Local Manager. These applications embody the modular reuse of the network communications infrastructure provided by the SSM, which handles network communications using protocols described in the SPA standards. Additional software modules can be added to the Infrastructure tier to handle new subnet types and different physical transmission media types. The SSM also provides a full API, reducing the development time and complexity of creating SPA standards compliant applications.

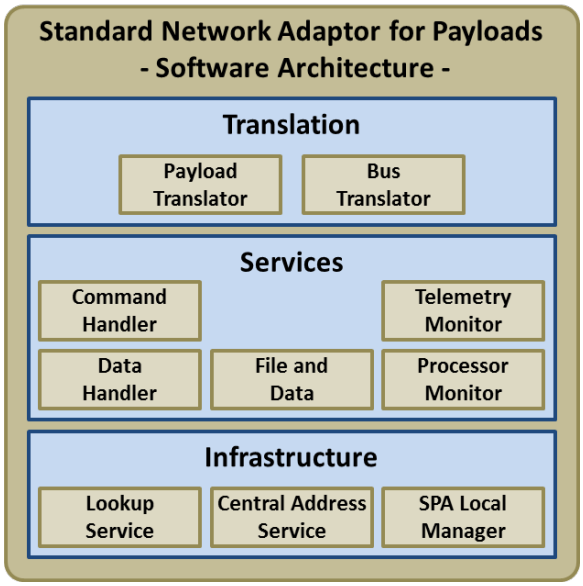


Figure 4.1. SNAP architecture

4.3 Services

There are five applications that make up the Services tier. These are SPA applications and provide services that are available to all payloads and busses. The applications are; Data Handler, File and Data Service, Command Handler, Telemetry Monitor, and Processor Monitor. These services are offered to payloads and busses, but their use is optional. This optional service design allows the flexibility to integrate payloads and busses of varying features and capabilities.

The optional services are provided in case there is a service gap that results from the payload and spacecraft being developed separately. Consider that currently a spacecraft and its payloads are designed together as a system, and system engineers provide a division of responsibility between the spacecraft and its payloads on a per system basis. If the payload and spacecraft are developed independently, then there could be responsibilities that are not provided by either. An example of this is data storage. On a spacecraft designed for one mission, it could be decided that the payload will handle its own data storage, and on a design for another mission it may be decided that data storage will be handled by the spacecraft. Since there is a potential shortfall of needed services SNAP has

to be capable of providing any service that the spacecraft and payload expects the other to provide. In the case of data storage, SNAP has a service application called the File and Data Service that can provide the data storage functionality. However, if either the spacecraft or payload already provide data storage, then using the File and Data Service application is not mandatory.

The software applications of the service tier do not need to be modified as different payloads and busses are integrated, but it is possible to add new applications for additional service functionality.

4.3.1 File and Data Service

File and Data Service is a SPA application that manages file and data buffers. Its primary function is to store data produced by the payload. File and Data Service provides a standard file interface that provides the ability to: create, delete, open, close, read, write, append, and list files. In addition to payload telemetry data, this application is capable of storing configuration files and data sent by any of the other service applications.

By managing file and data buffers, File and Data Service allows the integration of payloads to the spacecraft system with minimal impact to the bus. Use of this application is optional, as the system allows for storage of data on the payload hardware, or payload data being transmitted straight through SNAP to the spacecraft bus. File and Data Service is particularly useful however, in a situation where neither the payload nor the bus have the capability to store data, or an instance where the systems engineer chooses to keep the payload data separated from the bus. See the use case in Figure 4.2.

4.3.2 Data Handler

This application acts as a throttle for dispensing data that is stored by the File and Data Service. Architecturally, the Data Handler separates the concerns of the rate of data reporting from the concerns of file storage. The Data Handler reduces both dropped packets and impact to the bus by controlling the rate at which payload and SNAP telemetry data

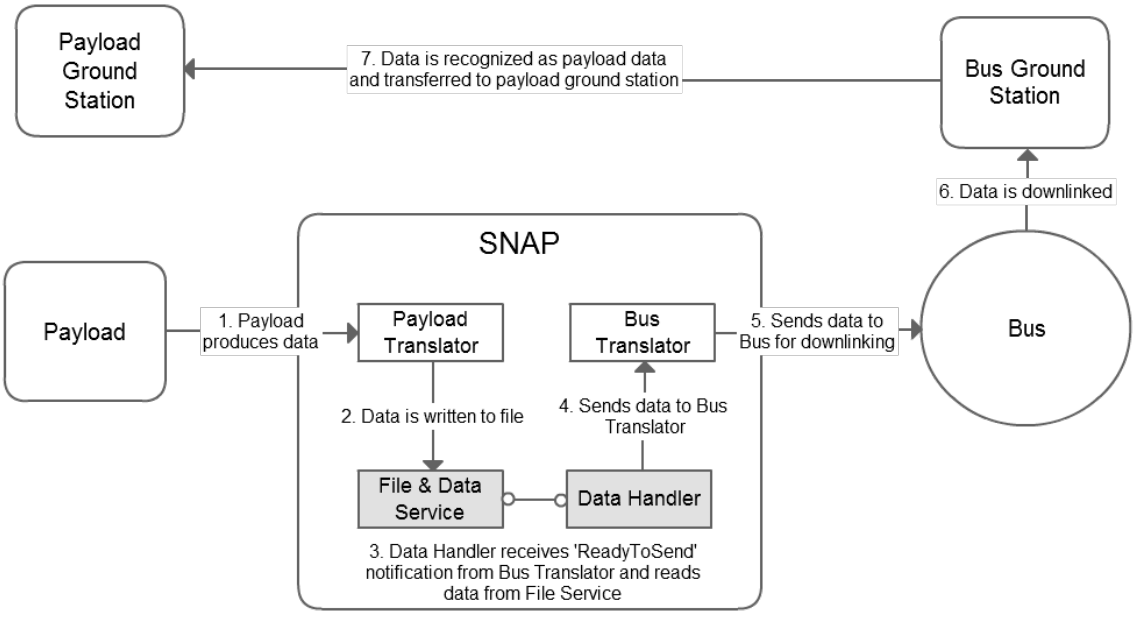


Figure 4.2. Payload sends data to ground via File and Data Service

is passed to the spacecraft bus. Data packets are not sent until the Data Handler receives a notification from the spacecraft bus that the bus is ready for data.

The flow of data is orchestrated as follows: File and Data Service sends a notification containing file meta-data to the Data Handler each time a file is created or updated. This file information is stored in a priority queue that is serviced when the spacecraft bus alerts the Data Handler to send more data. The Data Handler then uses the file information from the priority queue to request data from the File and Data Service. The role of the Data Handler in the SNAP software is also illustrated in Figure 4.2.

The Data Handler was designed to simplify synchronization of system data flow. This makes it easier to integrate payloads and busses that have different data rates.

4.3.3 Command Handler

The Command Handler manages system command messages. This a very powerful SPA application that has the flexibility to be used in a number of different ways. The Command Handler features a real-time execution queue which orders commands based on

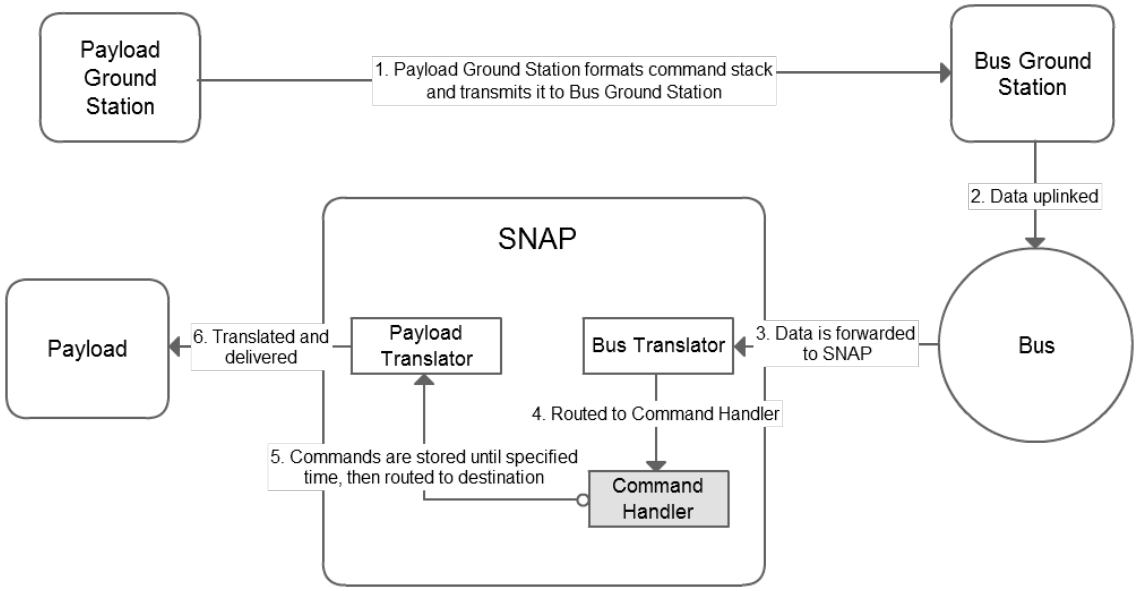


Figure 4.3. Ground sends timed command stack

time (seconds) and a priority field. Command execution can be tailored to a specific system by adjusting the rate that the Command Handler releases the commands from the queue. For example; the Command Handler can be configured so that once per second, n number of commands are released. These released commands are encapsulated in messages that may be destined for the payload, any SNAP component, or the spacecraft bus.

This application also features the ability to store a number of command sequences. These stored command sequences allow the Command Handler to be used to execute timed system routines. The real-time execution queue used in combination with the stored command sequence structures provides the means for commands to be stored, delivered and executed at a specific time, and reused. For example, a command could be sent from the ground station: At system time x, load command sequence b into the real-time execution queue. The last command of sequence b could be to load another sequence. The Command Handler is an application that is intended to be used in creative ways; even beyond what was initially imagined by its designers. See Figure 4.3 for the primary use case for the Command Handler.

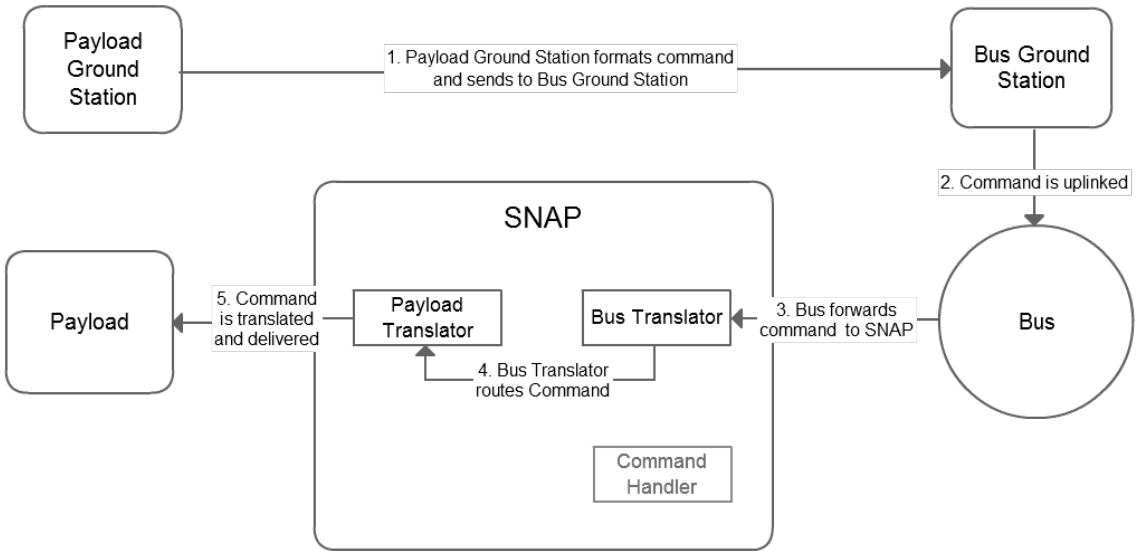


Figure 4.4. Ground sends command to payload

Although the Command Handler may be a very powerful tool, its use is also optional. There may be situations where the payload ground station would like to send a command directly to the payload for immediate execution. The SNAP software allows for this, as encapsulated command messages can be routed up through the bus, then translated by SNAP and delivered to the payload. This use case is shown in Figure 4.4.

4.3.4 Processor Monitor

Another application in the service tier of the SNAP software is the Processor Monitor. This application publishes state of health and other telemetry specific to the SNAP system. This application should be customized for each SNAP instantiation.

4.3.5 Telemetry Monitor

The Telemetry Monitor is a generic SPA application that manages telemetry data produced by other SNAP components. The application subscribes to the telemetry notifications published by any of the hardware or software components of the system. These subscriptions can be set up based on a configuration file or by system query. The Telemetry

Monitor coordinates subscriptions and marks stored files as telemetry. It has the potential to contain logic to trigger events based on observed telemetry conditions which could be used to have the system perform service tasks autonomously. The Telemetry Monitor is not responsible for storing or sending data as these are done by the File and Data Service, and Data Handler applications, respectively.

4.4 Translation

Translation between different hardware and software interfaces is at the heart of SNAP. The software tier that represents translation contains the SNAP software modules that are specific to the attached payload and spacecraft bus. These applications perform the function of translating device communication protocols to the standard SNAP formats and protocols, and so are individually developed on a case by case basis for each payload and each spacecraft bus. The good news is; since a translator application is the single part of the software that adapts a device to the SNAP system, it only needs to be done a single time. In other words, once a bus translator application is written for a particular spacecraft bus, the integration of that bus to the SNAP software is complete and remains unchanged as any number of payloads are adapted to the system. Likewise, once a payload translator application adapts a particular payload to SNAP, that application remains the same as a host spacecraft bus is adapted to SNAP.

4.4.1 Bus Translator

Interpreting communications between a spacecraft bus and SNAP is done by a Bus-Translator application. It is responsible for receiving and handling messages coming from the spacecraft side of SNAP. These messages may be information from the bus itself or encapsulated commands from the ground station intended for SNAP or the payload. The Bus Translator application must also translate SNAP messages into something the spacecraft bus can handle. Messages being sent from the Bus Translator to the bus need to be in the bus protocol, and may include requests directly to the bus or encapsulated telemetry data on its way to the payload ground station.

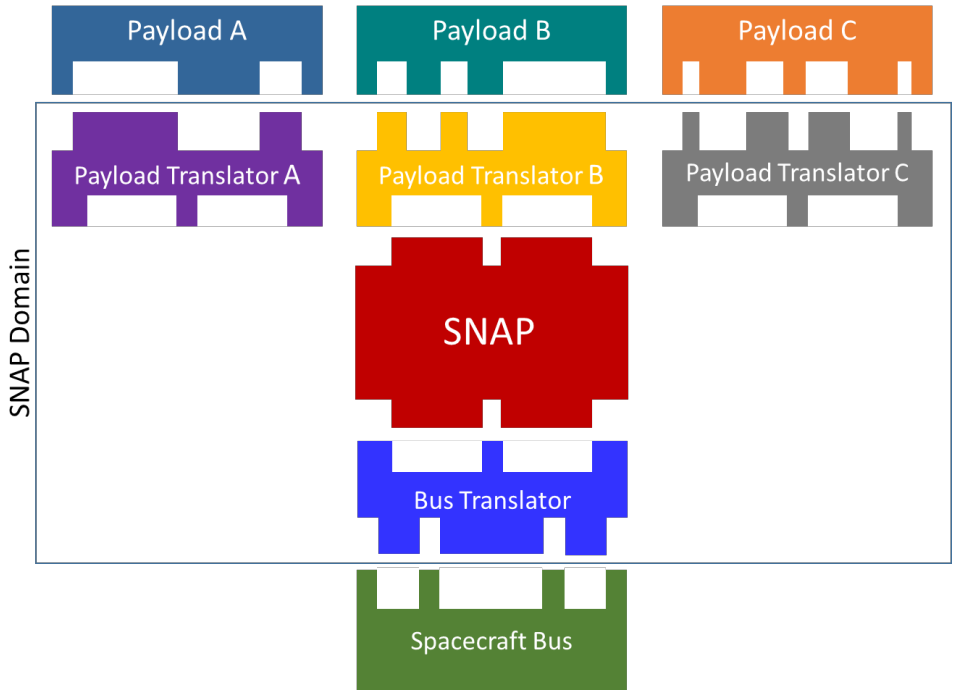


Figure 4.5. A spacecraft bus is adapted to SNAP a single time and is then compatible with any number of SNAP-adapted payloads

Figure 4.6 shows a use case for the Bus Translator. This use case depicts a situation where the payloads ground station needs the spacecraft to change attitude (slew) prior the payload being commanded to perform a task.

To make the task of adapting a new spacecraft bus easier, SNAP provides a generic Bus Translator which includes as much of this application as possible. The application contains roughly half of the needed functionality by implementing the SPA interface which is used for communication between the Bus Translator and the rest of SNAP. The part of the application that cannot be provided ahead of time is the custom portion for a specific spacecraft bus. Using this schema, the complexity of adapting a specific spacecraft is reduced, and spacecraft that are not yet invented can be adapted to SNAP in the future. Additional information about translator applications for spacecraft busses can be found in section 3.1 Integrating a Spacecraft Bus.

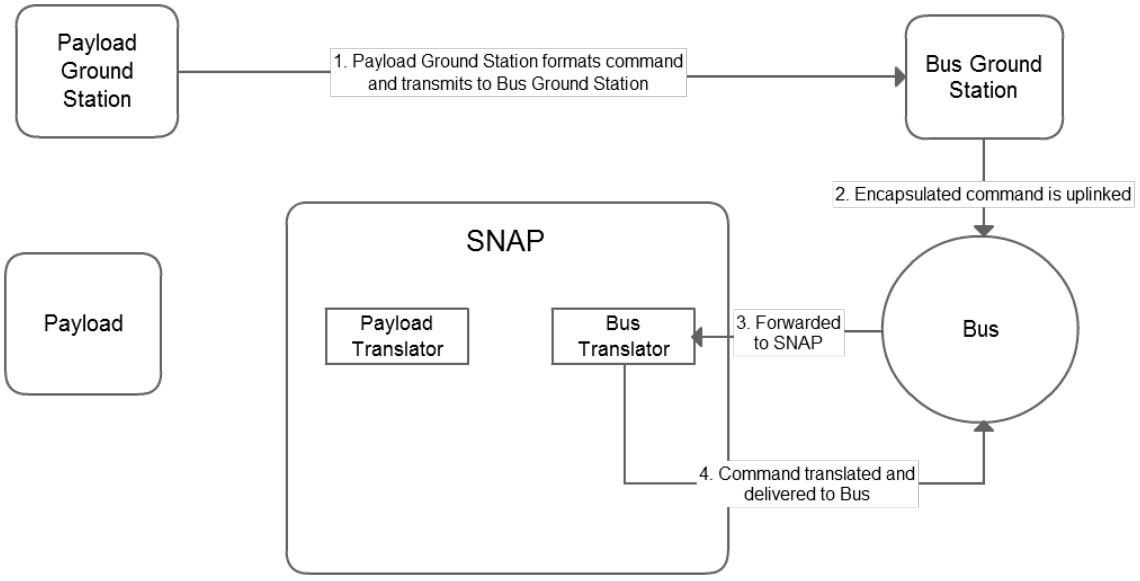


Figure 4.6. The Bus Translator handles command destined for spacecraft bus

4.4.2 Payload Translator

A Payload Translator application is responsible for interpreting communications between a payload and the SNAP software. The application must accept messages from SNAP and convert those messages into the payload protocol before sending those messages to the payload. The Payload Translator must also accept messages or data from the payload, convert the data from the payload protocol to the SNAP protocol, and then perform the appropriate operation including sending the message to the correct party in the SNAP software.

A use case involving both translator applications is shown in Figure 4.7. In this situation, the payload needs to know the position of the spacecraft. The figure depicts how the Payload Translator and Bus Translator applications facilitate communications between the spacecraft and the incompatible payload it is hosting.

Again, to reduce development time and duplication of effort, SNAP provides a generic Payload Translator for use in developing an application for a specific payload. The generic translator contains the SPA interface portion for communication between a Payload Translator application and the rest of the SNAP applications. The part of the application that

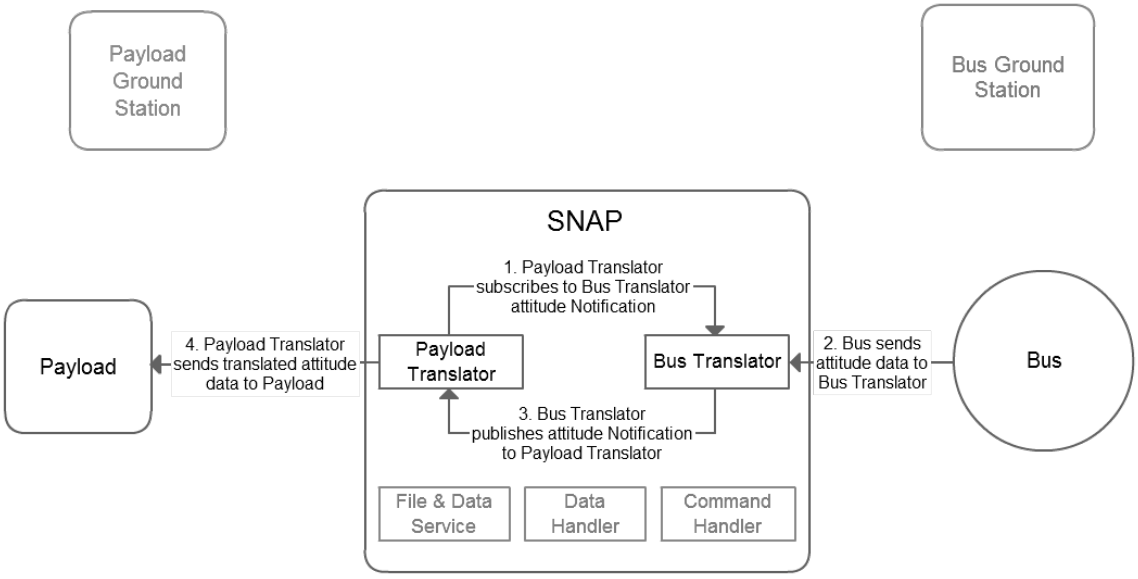


Figure 4.7. The Payload Translator requests data from spacecraft bus

is not provided ahead of time is the portion that must be customized for a specific payload. This design choice of creating a framework in which a developer completes the application for payload eliminates the need for SNAP to provide an implementation for every possible payload in existence, and allows payloads that are not yet invented to be more easily adapted to SNAP in the future. Additional information about translator applications for payloads can be found in 5.2 Integrating a Payload.

CHAPTER 5

SYSTEM INTEGRATION

5.1 Integrating a Spacecraft Bus

A spacecraft bus is adapted to SNAP by implementing a bus translator application. Implementing a translator application for a spacecraft is a one-time effort; once the spacecraft is adapted to SNAP, it is compatible with any number of SNAP-adapted payloads. To facilitate developing the translator application for a specific spacecraft bus, SNAP provides an abstract base class named BusTranslator. The abstract base class already contains the SPA interface code for communication between a bus translator application and the rest of the SNAP applications. Providing this functionality reduces duplication of effort and development time for individual spacecraft busses.

To complete the integration, a developer derives an application from the provided base class. A graphical representation of the bus translator showing the division of what is provided and what is left to be implemented can be seen in Figure 5.1. The developer must implement the abstract functions appropriately for the spacecraft bus that is being integrated. Implementing these functions is the essence of translating the data from SNAP (which uses SPA protocol standards) into the bus protocol and sending the data to the bus. The application must also be written to convert data from the bus protocol into SPA protocol and call the appropriate setter function already implemented by the non-abstract portion of the provided application. This represents translation of messages coming from the bus on their way to SNAP or the Payload. Additional implementation details about bus translator applications are provided in 6.1.

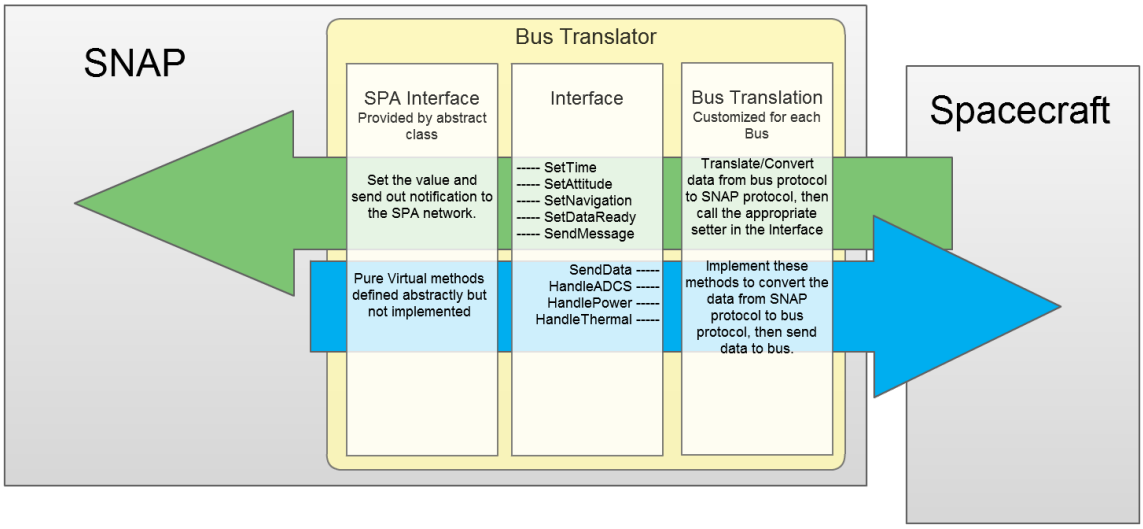


Figure 5.1. Spacecraft bus translation interface

5.2 Integrating a Payload

A payload is adapted to SNAP by implementing a payload translator application. Implementing this translator application for a payload is a non-recurring effort. The payload is adapted to SNAP a single time and is then compatible with any number of SNAP-adapted spacecraft. SNAP provides an abstract base class named PayloadTranslator to facilitate developing the translator application for a specific payload. The abstract base class already contains the SPA interface code for communication between a payload translator application and the rest of the SNAP applications. Providing this functionality in the base class reduces duplication of effort and development time for each individual payload.

To complete the integration of a payload to SNAP, a developer makes a SPA application that is derived from the base PayloadTranslator. A graphical representation of the payload translator can be seen in Figure 5.2, showing the division of what is provided and what is left to be implemented. The developer must implement the abstract functions according to the particular payload they are integrating. These implemented methods will be called by functions in the non-abstract side of the application, and represent the translation of SPA standard messages going in the direction of the payload. If a payload cannot provide the functionality represented by any of these abstract functions, the developer may implement

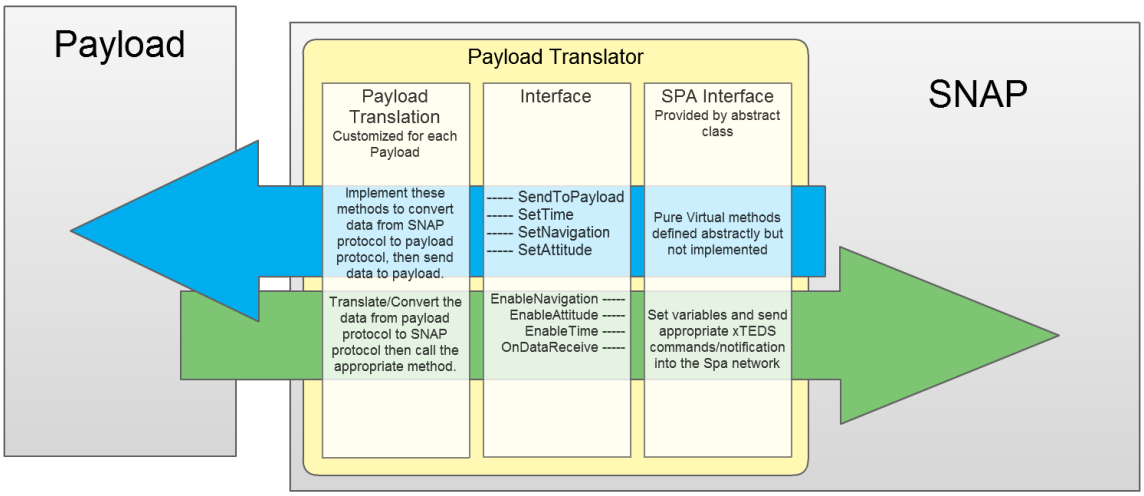


Figure 5.2. Payload translation interface

a function stub. The developer then uses the payload protocol to receive messages and make appropriate calls to the non-abstract functions provided in the base class. This represents translation of payload communications going in the direction of SNAP. Additional information about bus translator applications is provided in 6.2.

CHAPTER 6

SNAP IMPLEMENTATION

6.1 Spacecraft Bus Translator Implementation

This section takes a closer look at implementing a bus translator application which is needed to adapt a spacecraft bus to SNAP. The application is derived from the provided C++ abstract base class named BusTranslator, and will be run on the same processor unit as the rest of the SNAP applications. The abstract base class is in fact a SPA Application and contains the SPA interface code for communication between a bus translator application and the rest of the SNAP applications. SPA Applications use the SSM network infrastructure to communicate with other system components using messages and protocols defined by the SPA standards.

Implementing a bus translator application for a spacecraft is a non-recurring effort. The spacecraft is adapted to SNAP a single time and is then compatible with SNAP-adapted payloads. To finish the integration of a spacecraft to SNAP, the derived bus translator application is completed by implementing the abstract functions according to the spacecraft bus that is being integrated. These implemented methods will be called from methods in the non-abstract portion of the application, and represent the translation of SPA standard messages going in the direction of the spacecraft. If a payload cannot provide the functionality represented by any of these abstract functions, the developer may implement a function stub. The application must also be written to convert data from the bus protocol into SPA protocol and call the appropriate setter functions already implemented by the non-abstract portion of the provided base application. This represents translation of messages coming from the bus on their way to SNAP or the Payload.

The UML diagram in Figure 6.1 depicts the inheritance hierarchy for bus translators.

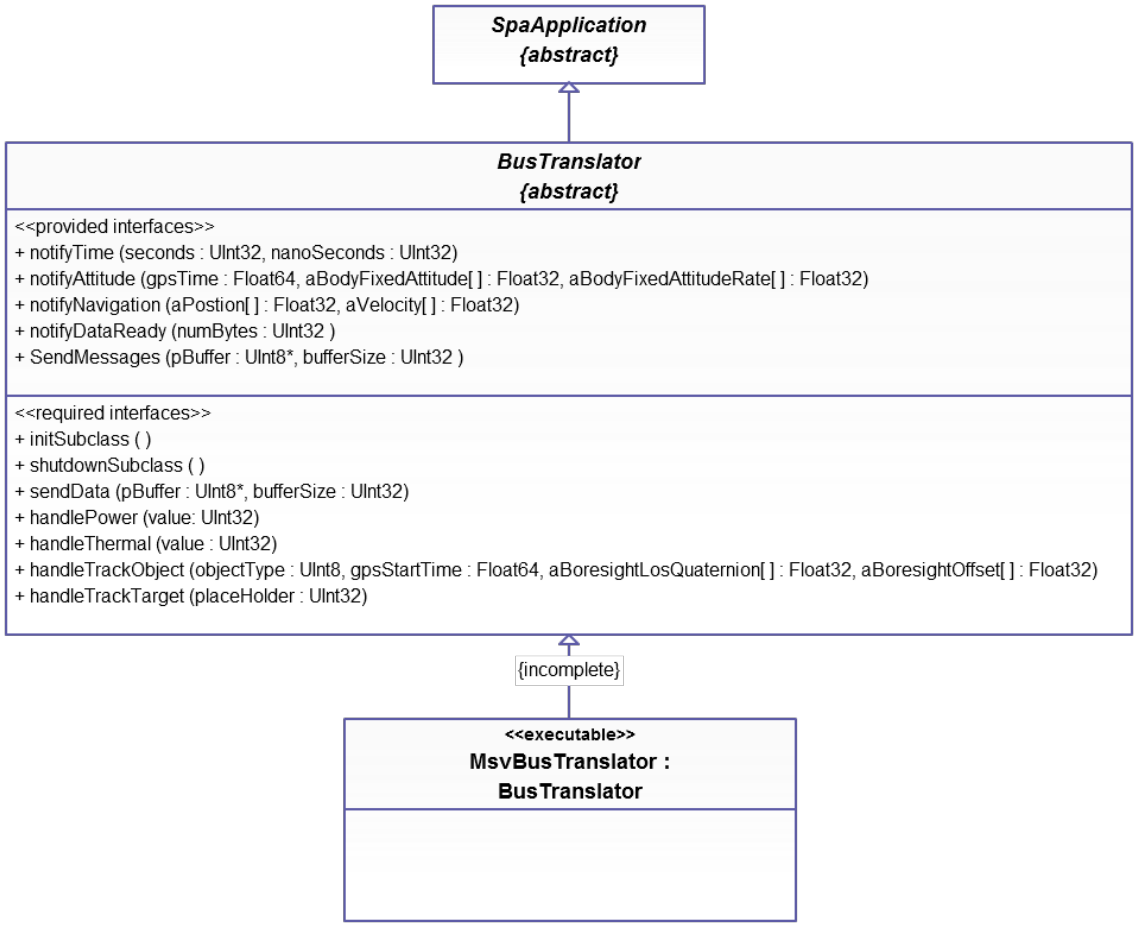


Figure 6.1. Each derived bus translator must implement the pure virtual functions of the parent bus translator

The diagram also shows the specifications for the required interfaces (pure virtual functions) that must be implemented by each derived bus translator.

6.1.1 Provided Functionality

In addition to the functionality provided by the C++ class SpaApplication, the base class BusTranslator provides five public functions designed specifically to be called by the derived bus translator applications as data is received and translated from the spacecraft bus.

The SendMessage function is provided to allow a derived bus translator application

to pass received spacecraft data to the SNAP software. The function has a parameter list consisting of a pointer to a (byte) buffer, and an unsigned 32 bit integer representing the size of the data buffer.

There are four notify functions provided as a way to issue data notification messages to SNAP when the derived bus translator receives data published by the spacecraft. There is one function each for time, attitude, and navigation data. The notifyDataReady function is called by the derived bus translator to let SNAP know that the spacecraft is ready for more data.

6.1.2 Required Implementation

The C++ abstract base class BusTranslator has seven functions that must be implemented by all derived bus translator applications. These functions are defined in BusTranslator.hpp as pure virtual so that the derived bus translator will not successfully compile until the functions are implemented. These seven function prototypes have no return value, so if any of the functions are not necessary to adapt a spacecraft to SNAP, they may be implemented as stubs, or empty functions.

The initSubclass and shutdownSubclass functions may be written to perform initialization tasks and tasks in preparation for shutting down. For example, a developer can use the initSubclass function to set up buffers and start threads. The shutdownSubclass function can be used to join threads and perform any other necessary clean-up.

The sendData function is called by the base BusTranslator when there is data coming through the SNAP software that is to be forwarded to the spacecraft bus. The parameters are a pointer to a (byte) buffer and an unsigned 32 bit integer representing the size of the data buffer. The function is intended to be used by a developer to both prepare and send the data to the spacecraft bus. Depending on the specific spacecraft and the over-all system configuration, preparing the data could represent anything from a straight pass-through of data, to extensive translation and processing of the data. In this function, the developer has the responsibility to get the data to the spacecraft bus in a format the bus understands. Exactly what this entails and how it is implemented is the decision of the

developer integrating the spacecraft.

There are four handler functions to implement that may be used to request services from the spacecraft bus. These functions are called by the base BusTranslator when the application receives commands to affect the power, thermal, or ADCS state of the spacecraft. The implementation of these functions is intended to be used to then request these services from the spacecraft using the spacecrafts native protocols. In other words, these functions can be implemented to translate service requests from the incoming SPA message protocol into the protocol of the spacecraft, then pass the requests to the spacecraft.

6.2 Payload Translator Implementation

This section takes a closer look at implementing a payload translator application which is needed to adapt a payload to SNAP. The application is derived from the included C++ abstract base class PayloadTranslator. The derived application will be run on the same processor unit as the rest of the SNAP applications. The abstract base class is a SPA Application and contains the SPA interface code for communication between a payload translator application and the rest of the SNAP applications.

Implementing a translator application for a payload is a non-recurring effort. The payload is adapted to SNAP a single time and is then compatible with SNAP-adapted spacecraft. To complete the integration of a payload to SNAP, a developer completes a derived payload translator application by implementing the abstract functions according to the particular payload they are integrating. These implemented methods will be called by functions in the non-abstract side of the application, and represent the translation of SPA standard messages going in the direction of the payload. If a payload cannot provide the functionality represented by any of these abstract functions, the developer may implement a function stub. The developer then uses the payload protocol to receive payload data and make appropriate calls to the non-abstract functions provided in the base class. This represents translation of payload communications going in the direction of SNAP.

The inheritance hierarchy for payload translators is depicted in the UML diagramin Figure 6.2. The diagram also shows thespecifications for the required interfaces (pure virtual

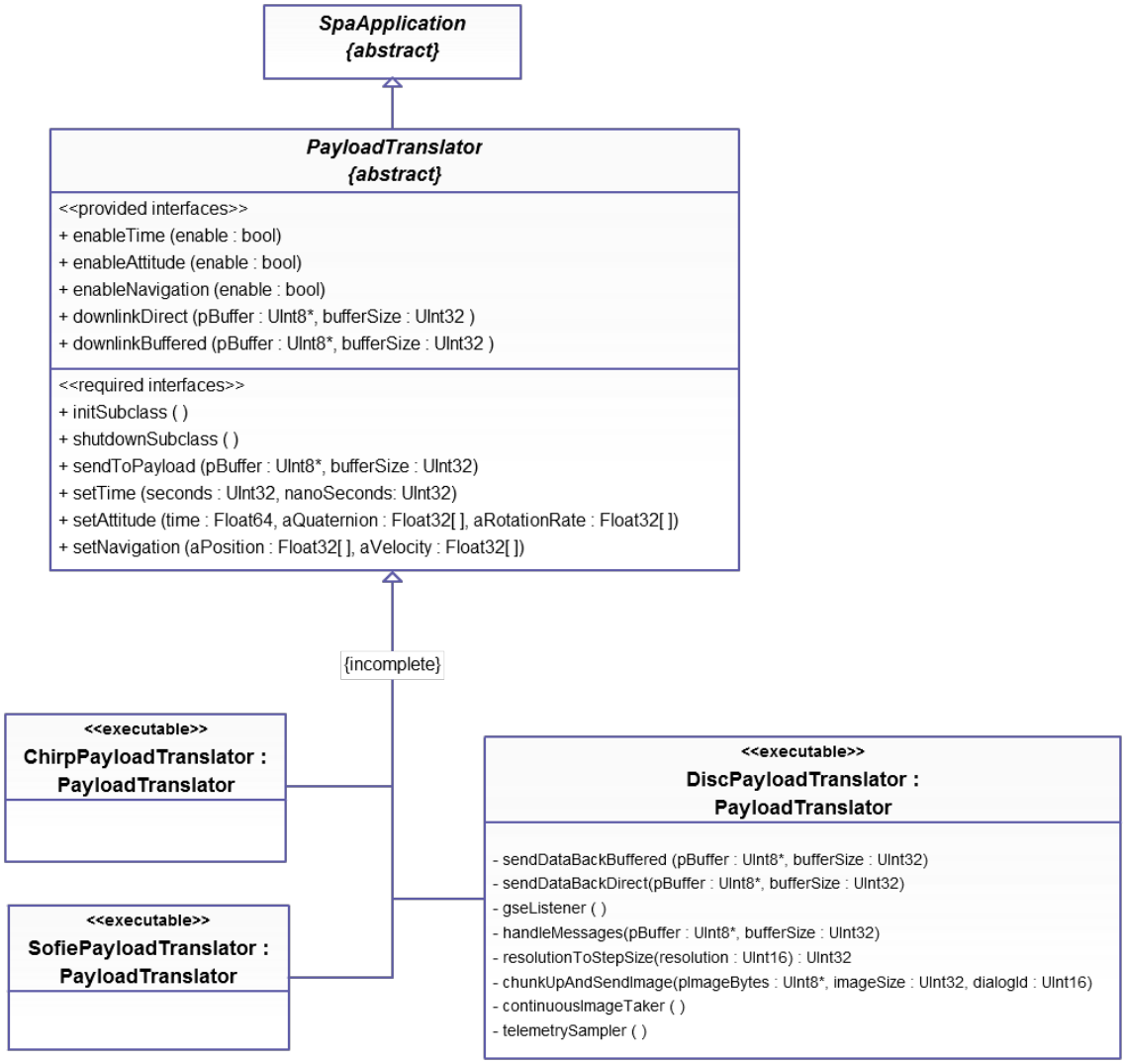


Figure 6.2. Each derived payload translator must implement the pure virtual functions of the parent payload translator

functions) that must be implemented by each derived payload translator.

6.2.1 Provided Functionality

In addition to the functionality provided by the C++ class SpaApplication, the base class PayloadTranslator provides five public functions designed specifically to be called by the derived payload translator applications.

There are three enabler functions provided, one each for time, attitude, and navigation.

These functions are used (called) by derived payload translators to enable or disable the flow of time, attitude, and navigation data notification messages coming from SNAP. These functions have no return value, and each have a boolean parameter. Use true to enable, and false to disable the notifications.

Two functions are provided to allow a derived payload translator to pass received payload data to the SNAP software. The two functions have identical parameter lists consisting of a pointer to a (byte) buffer, and an unsigned 32 bit integer representing the size of the data buffer. Call downlinkBuffered to append this data to a file managed by SNAPs File and Data Service. The function downlinkDirect is used to send data directly if the SNAP hardware is configured with a downlink transmitter. These functions have no return value.

6.2.2 Required Implementation

The C++ abstract base class PayloadTranslator has six functions that must be implemented by all derived payload translator applications. These functions are defined in PayloadTranslator.hpp as pure virtual so that the derived payload translator will not successfully compile until the functions are implemented. These six function prototypes have no return value, so if any of the functions are not necessary to adapt a payload to SNAP, they may be implemented as stubs, or empty functions. The initSubclass and shutdownSubclass functions may be written to perform initialization tasks and preparatory tasks for shutting down. For example, a developer can use the initSubclass function to set up buffers and start threads. The shutdownSubclass function can be used to join threads and perform any other necessary clean-up.

The sendToPayload function is called by the base PayloadTranslator when there is a message coming through the SNAP software destined for the payload. The parameters are a pointer to a (byte) buffer and an unsigned 32 bit integer representing the size of the data buffer. The function is intended to be used by a developer to both prepare and send the data to the payload. Depending on the specific payload and the over-all system configuration, preparing the data could represent anything from a straight pass-through of data to the

payload, to extensive translation and processing of the data. In this function, the developer has the responsibility to get the data to the payload in a format the payload understands. Exactly what this entails and how it is implemented is the decision of the developer integrating the payload. An example implementation can be seen in the DiscPayloadTranslator.cpp file.

There are three setter functions; time, attitude, and navigation. These functions are called when the payloadTranslator receives the corresponding time, attitude, or navigation notification message from SNAP. These three setter functions can be implemented to pass time, attitude, and navigation data to the payload.

CHAPTER 7

CONCLUSION

7.1 Results

The SNAP software provides an adaptable interface for hosted payloads or other natively incompatible spacecraft and payload. The derived bus translator represented in Figure 6.1, (MsvBusTranslator) was completed in the SNAP source code, and was used to adapt Northrop Grummans Modular Space Vehicle (MSV) to the SNAP system during the demonstration of SNAP software conducted November 20, 2013 for SMC/XR.

The demonstration incorporated three payloads which were selected to represent different payload types, protocols, and data rates. The three derived payload translators represented in Figure 6.2, (SofiePayloadTranslator, DiscPayloadTranslator, and ChirpPayloadTranslator) were used to adapt the payloads to MSV.

The SofiePayloadTranslator application communicated over an RS-422 link with a low data-rate (115kbps) simulation of SDLs Solar Occultation For Ice Experiment (SOFIE) sun sensor. Actual on-orbit data from SOFIE was used for the simulation.

The DiscPayloadTranslator application used a SpaceWire interface to communicate with a Digital Imaging Space Camera (DISC) imager. The imager is an engineering unit similar to the DISC imager used on the International Space Station. The imager was used to demonstrate real-time command and control of a payload, and sent image data through SpaceWire at a medium data-rate (40 Mbps).

The third payload was adapted to the SNAP system using the ChirpPayloadTranslator application. This payload was a simulation of a next-generation Commercially Hosted Infrared Payload (CHIRP), and featured high rate data (141 Mbps) over SpaceWire.

These three payloads were integrated in three days, which is a dramatic improvement

over the traditional three to six months that it would normally take. These results and the SNAP message were shared in two related conference papers presented by SNAP team members [5] [7].

The successful demonstration of SNAP and its modular concepts received largely positive feedback from the attending government and industry representatives. Interest in SNAP was sufficient to follow up the demo with an industry workshop that was conducted in early 2014.

REFERENCES

[1] Office of the Under Secretary of Defense, "Better buying power 2.0: Continuing the pursuit for greater efficiency and productivity in defense spending," in *Acquisition, Technology and Logistics Memorandum*, November 2012.

[2] Deputy Secretary of Defense, "Operation of the defense acquisition system," in *Instruction 5000.02*, November 2013.

[3] M. Gruss, "Shelton: 'we've got to start now' on milspace reforms," *Space News*, vol. 24, April 2013, issue 49, page 14.

[4] United States Government Accountability Office, "Defense acquisitions: Dod efforts to adopt open systems for its unmanned aircraft systems have progressed slowly," in *GAO-13-651 Report to the Subcommittee on Tactical Air and Land Forces Committee on Armed Services House of Representatives*, Month Year.

[5] R. Ewart and G. Ellis, "Mona framework for leading change: The small satellite paradigm," in *Proceedings of the AIAA/USU Conference on Small Satellites*, August 2014, paper Number SSC14-V-5.

[6] C. Salmon, "Standard network adapter for payloads (snap) software information guide," in *United States Air Force Contract No. FA8814-09-D-0001, Task Order 1, Subtask SNAP*, 2014, document Number SDL14-049.

[7] G. Ellis, P. Graven, Q. Young, and J. Christensen, "Standard network adapter for payloads (snap)," in *Aerospace Conference, 2014 IEEE*, March 2014, pp. 1–8.

[8] J. Christensen, D. Anderson, M. Greenman, and B. Hansen, "Scalable network approach for the space plug-and-play architecture," in *Aerospace Conference, 2012 IEEE*, March 2012, pp. 1–10.

[9] American Institute of Aeronautics and Astronautics, "Space plug-and-play architecture," in *AIAA Standards*, November 2013.

[10] International Organization for Standardization, "Information technology open systems interconnection basic reference model: The basic model," in *ISO/IEC 7498-1:1994(E)*, 1994.